Informing content-driven design of computer programming games: a problems analysis and a game review

Lieve Laporte
Mintlab – KU Leuven
Leuven, Belgium
lieve.laporte@kuleuven.be

Bieke Zaman
Mintlab – KU Leuven,
Leuven, Belgium
bieke.zaman@kuleuven.be

ABSTRACT
Currently, educational games are being developed to teach children the basics of computer programming. Research and design of such games is usually based on general learning theories. Yet, computer programming poses specific types of difficulties to novice programmers. Taking into account these particular characteristics and problems of computer programming as a learning content in the design of programming games could allow for producing games that are more suitable to the needs of novice programmers. This paper first reports on a novice programmer problems analysis, to gain insight into learners' specific difficulties. Then, a review of existing programming games is presented to investigate how and to which extent these games deal with specific programming problems. The results of these studies aim to contribute to the requirements and ideation phases of a programming game design process, thereby informing a learning content-driven design perspective.

Author Keywords
Educational games; computer programming games; instructional design.

ACM Classification Keywords
H.5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous; K.8.0. Personal computing: General.

INTRODUCTION
Computer programming is currently considered a new literacy. Although opinion remains divided on the extent to which people should be programmers - i.e. "everybody should learn to code" versus the dangers of people trying to teach themselves the technology du jour [41] -, programming is generally believed to increase critical-thinking and problem-solving abilities and teach core concepts of how things work in today's society [51]. As a consequence, it is frequently argued that programming should be incorporated into the curriculum at a younger age [51]. Because of this challenge, new ways of learning how to code, such as the use of educational games, have been suggested [22][45].

Educational games have been found to engage young learners in active and behavioural learning [24][44], facilitated through providing challenge, support and feedback, and they encourage learners to solve problems through the exploration of simulated environments [16]. In addition, games can be tailored to individual ability levels [42], and by capturing people's attention, they can increase the time spent on learning through repetition and practice [34].

The design of educational games is mostly approached from a general, holistic learning theory perspective. Paradigms or theories such as constructionism or experiential learning are selected as a pedagogical framework for integrating *any learning content* into games. However, researchers in various learning domains have recognized the need for *content-specific* guidance in the design of educational games. For example, Echeverria et al. claimed that there is no structured methodology available to guide the design of games to teach conceptual physics [14]. Similarly, Plass et al. have argued that learning activities from specific domains should be conveyed through content-specific mechanics in a game [33].

Translating these concerns to the field of programming education, we aim to investigate which specific programming problems novice learners experience, and how these are addressed within existing programming games. In addition, we consider if and how the results of these investigations could inform a content-specific programming game-design approach.

RELATED WORK

Programming problems
Computer programming and computational thinking (i.e., the basic reasoning and problem-solving required for learning how to program) encompass a broad range of constituent knowledge and skills [50], which are known to be difficult, very particular and atypical skills in a learner's curriculum [43]. Novice programmers' knowledge is often

NordiCHI '16, October 23-27, 2016, Gothenburg, Sweden
 ISBN 978-1-4503-4763-1/16/10…$15.00
DOI: http://dx.doi.org/10.1145/2971485.2971499

referred to as 'fragile', i.e. knowledge that is missing, inert, or misused [26] [31]. Indeed, to a novice, the concept of programming is something completely new, not really like anything they have learned before. Although there is still a point of contact between the knowledge learners currently have and the new material they have to learn (e.g. similarities with writing lists of instruction for people to follow [40]), applying these analogies from other domains to the new task of programming can also be misleading [4]. For example, natural language is a potential source of misconceptions because of the discrepancies between word meanings in natural language and their use in programming [35]. In addition, the level of description required, the explicit control structures, the very particular syntax rules [17] and the variety of programming constructs [23] [27] [46] introduce a number of specific difficulties. Finally, novice programmers have been found to have problems with both designing [26] and testing their programs [31].

Programming game design research

Exploring whether these specific characteristics and difficulties are being dealt with nowadays, we found that previous research has approached programming game design from different perspectives. For example, some studies have taken an efficiency perspective by investigating the learning outcomes of programming games (e.g. [2][7]). Other studies focused on technocentric questions, such as the adaptability of programming games to different programming languages, a maximum of teaching contexts (e.g. [29][30]) or on the implementation of a variety of programming constructs (e.g. [32]). Still other studies took a more explicit game design perspective. They zoomed in on the design of specific game details, or considered a game's learning content from the perspective of the learning theory that was at the base of the game design. For example, Barnes et al. discussed the role of rewards for completion of programming tasks, and the balance between programming and playing in the game [1]. Esper et al. addressed the design of quests that provide scaffolding to support students in learning [15]. Kazimoglu et al. developed a game framework that allows for the development of five core computational thinking skills [19]. Although these studies presented a variety of research topics, they did not consider a bottom-up learning content perspective. In other words, it is not clear how the games resulting from this work would take into account the particularities of the subject matter, i.e. how they would address specific and typical problems with acquiring programming skills.

A content-driven design perspective

The specific and atypical difficulties of computer programming as a learning content, rather unremarked in previous programming game research, make the case for the exploration of a bottom-up, learning-content-driven (or shorter, content-driven) design perspective, as has also been suggested by researchers in other learning domains [14] [33]. More specifically, it is worth asking whether research on the design of (content-related aspects of) programming games should confine itself to design decisions based on general learning principles, envisaged for any learning content. To fully reach the educational potential of programming games, an additional and explicit focus on the specificity of programming problems could be beneficial as a starting point for design.

In this paper, we propose three subsequent steps in the initiation of such a learning-content-driven approach to the design of programming games. First, we compiled programming problems (Study 1), to gain insight into the specificity of difficulties experienced by novice programmers. Second, we analysed a set of 19 existing programming games (Study 2) to determine if and to which extent they addressed the problems identified in Study 1. Third, we integrated the findings of both studies, in order to, eventually, inform the analysis and ideation phases of the programming game design process from a specific learning-content perspective. The remainder of this paper is structured according to this three-fold bottom-up approach, i.e. we start with a data-driven description of Study 1 and Study 2, followed by an interpretative integration of the results of both studies, which then provides a foundation for discussing game design and research implications.

STUDY 1: PROBLEMS ANALYSIS

To collect novice programmer problems, we first conducted a literature analysis, starting from the comprehensive reviews of Robins et al. [35] and Soloway & Spohrer [43]. However, we found that previous research mainly focused on novice programmers aged 18 and older. Given our goal of identifying concrete programming problems in the context of game design -for which the main target audience presumably consists of children younger than 18- we organized an expert workshop to collect 'young novice programmer' problems, occurring in everyday practice with children.

Participants

Fourteen CoderDojo instructors and ICT teachers were recruited to participate to our expert workshop. These participants all had experience with teaching computer programming to children between 7 and 18 years old.

Procedure

Participants were brought together, and the goals of the workshop were explained. Participants were asked to individually write down novice programmer problems from their experience in everyday practice. While doing this, they were instructed to be as specific as possible and provide examples. Each participant's ideas were then passed to a next participant, who could elaborate on these ideas, or provide new ones. We organized six rounds of this problem generation phase. Afterwards, participants discussed all ideas together, and refined and elaborated them. The entire discussion was recorded with a video camera and a digital voice recorder for further analysis.

Analysis

All novice programmer problems were listed together and similar problems were grouped. These groups were then categorized, starting from a programming framework from literature (i.e. categories as a result of an extensive review of novice programming problems by Robins et al. [35]). Next, each problem category was fleshed out with a summary and examples of the workshop findings.

Results

We identified 10 base categories of novice programmer problems, which were grouped into three main component groups. The first group, **process components**, consists of problems related to the basic steps in a programming process, i.e. *design*, *implementation of programming constructs*, and *evaluation*. The second group consists of underlying **cognitive components**, prerequisite to starting and completing a programming process, i.e. writing *syntax*, developing a basic *knowledge*, *mental models*, and *mapping* and *problem-solving* strategies. The third group consists of **affective components** supporting the programming process, i.e. *attitude* and *motivation*.

PROCESS COMPONENTS

Design of solution – The workshop results showed that young novices often don't know how to get started: they understand the assignment, but have no idea of the approach to take towards the solution. They find it difficult to split up the goal described in the assignment into smaller pieces and to structure basic operations into schemas or plans. When learners do start working on a solution, they immediately proceed to the implementation, and lack to create a preliminary design or plan on paper.

Implementation of programming constructs - Findings from the workshop presented specific problems with various programming constructs. For example, young novices often use variables to (repeatedly) designate constants. The same type of problem occurs with iteration constructs, where young novices are likely to write many times the same commands instead of using loops or functions. As for loops and conditionals, the scope of a variable was said to be a difficult concept to grasp.

Evaluation of program - Novice programmers were found to be poor at tracking and tracing code. They spend very little time testing code and often get frustrated when, for example, a program doesn't work because of trivial mechanical problems such as syntax errors. They consider diagnosing as the most difficult aspect of debugging, and tend to make the same mistakes over and over again.

COGNITIVE COMPONENTS

Syntax - From the workshop data we found that novice learning can become blocked by syntax errors. For example, non-native-English speakers often do not have enough knowledge of English. In addition, learners lack accuracy and precision, and they don't like to practice syntax. When a language transparent method is used, the mechanical syntax problems might be solved, but these methods tend to conceal the logic and reasoning of code and hampers learners' insights regarding correctness, analysis and efficiency. Hence, taking the step from these "visual" pseudo codes to "real" programming is difficult.

Knowledge - According to the workshop experts, young novice programmers lack knowledge of all the possibilities of code and programming structures and fail to spontaneously apply relevant prior knowledge. For example, they consider mathematics and logical thinking as part of other courses, and have to be given a lead before they are able to make use of this knowledge. As a consequence, they often lose themselves in unimportant details. A lack of knowledge can thus be the cause of many programming problems, but if too much theory is taught before applying it, children might lose interest and get lost in their overview of the code structure.

Mental models - In the workshop, it was stated that a basic knowledge of how a computer works is a prerequisite for being able to program. However, the correct functioning of a computer is so evident to young novice programmers, that having to actually program it feels unreal to them. This lack of mental model of how a computer works causes problems when programming. For example, young novices have difficulties in recognizing the difference between information that is either "not existing" or "not visible".

Problem-solving strategies – Learners were found to have difficulties conducting the logical reasoning to work out programming algorithms (e.g. the difference between *do while* and *do until*). In addition, they have problems with eliminating redundant information and selecting relevant information from a problem description, and with programming a solution step-by-step. They are often able to copy or apply prescribed problem-solving strategies, but when creating their own, they make use of general instead of problem- or programming-specific problem-solving strategies. As a consequence, they need really small step-by-step instructions to solve the problem, but the question is whether they learned something if these steps are so small.

Mapping – According to the workshop experts, children learn that things that are easy to comprehend are not always easy to describe in code or algorithms. For example, a child who wants to implement the movement of a bouncing ball in a physically correct manner might have problems to see how complex things really are. Learners have difficulties with translating real life behavior into code, i.e. with the level of description required by programming code to express the complex idea of a whole physical reality through partial programming functionalities.

AFFECTIVE COMPONENTS

Attitude - Besides problems directly related to programming, the workshop data also revealed attitude problems. More specifically, although children learn from

making trial-and-error mistakes, they sometimes have the attitude of trying and trying again, without thinking first. When searching for errors, they quickly tend to conclude that *it doesn't work* and lack the persistence to search for the causes of error messages. They are easily satisfied about a solution that accidentally works, which prevents them from learning from mistakes. In addition, the creativity required to improve the programming code is often missing. They frequently employ trial-and-error practices or quickly ask for correct answers when confronted with problems. Finding their own supporting information (for example, on the internet) appears to be difficult.

Motivation **-** The workshop findings also showed motivational problems. Especially very young novice programmers find it very boring to learn programming skills out of a textbook. When they have to program "for the sake of programming", they lack motivation for the project because they miss a specific, real and unique end product or goal, related to their own world.

STUDY 2: GAME REVIEW

We reviewed a set of existing programming games, assessing to which extent they address the preceding categories of novice programmer problems.

Games

We compiled an initial list of games to be analyzed, based on the games considered in Vahldick et al.'s [49] comprehensive programming games review. The latter authors searched academic and commercial databases, as well as (educational) game websites, thereby considering "all games used, proposed or developed to support introductory computer programming" from 2000 to 2013 [49, p.2]. We complemented the initial list with games launched in 2014 and 2015. Then, a final selection of 19 games was made by applying the following exclusion criteria to the initial list: (1) games that were not available for playing were excluded, because we believed that thorough analysis could only take place after at least partial play of a game; (2) outdated games, i.e. making use of obsolete programming languages or for which further development has been completely ended, were excluded. Names and references of the games on the final list are presented in the result and references sections.

Procedure and data analysis

The procedure of reviewing and analyzing the games consisted of a free play phase and an analysis phase. The free play phase was intended to become acquainted with both gameplay and learning content; games were played to completion whenever possible. During the analysis phase, we examined if and how the games addressed the novice programmer problems. Each game was played for a second time, and instances of (not) addressing each problem category were registered and described. We also scored each game on each problem category. We thereby made a distinction between three summative 'scores', representing the focus on and prevalence of a particular problem category: a '-' score indicating that no attention was given to the PC; a '+' score indicating a moderate focus (i.e. the game deals with the particular problem category in a superficial or fragmented fashion); and a '++' score indicating a strong focus (i.e. the game consistently and thoroughly deals with the problem category at hand).

Results

This section presents a description of how each problem category was dealt with in the programming games, illustrated with examples from specific games. Descriptions are organized based on the summative scores: for each problem category, we described how games with a (-), (+) and (++) score respectively approach that problem category.

Design of solution **(-)** We observed a few games presenting delineated, small-scale problems: since there is only one path to the correct solution, no preliminary design or planning is required. For example, Glitchspace requires learners to solve assignments in which they have to implement a short procedure. **(+)** The majority of games included moderate design or planning, mostly through assignments for which splitting up a problem in smaller pieces (e.g. dividing functionality into procedures), is required. For example, Lightbot is entirely constructed around a robot that has to be programmed to reach a target, by inserting sequences of commands into a main procedure and one or two sub procedures. Such procedure implementation problems of course need design and planning before actual implementation can start, but this kind of design is focused on aspects of a limited problem-solving process only, and not placed into the context of a more global challenge. **(++)** Only a few games presented problems for which a global and sometimes more abstract level of design and planning is required. For example, Codespells has a complex game world in which it is not readily clear which problems have to be solved, and which code would be appropriate to solve them. Kodable provides mazes, which can be solved in different ways. Learners have to look at the path structure of the presented problem and 'design' the optimal road before actually coding it. In these examples, design and planning is essential and in this sense, it is stimulated in these games. However, no support is provided to help learners start, evaluate and complete the design process.

Implementation of programming constructs **(-)** Our results presented some games that focus on a limited set of programming constructs, with no support for the specific difficulties of these constructs. For example, Robozzle presents nested functions and recursion problems, without support. **(+)** Some other games do focus on programming constructs that have been identified as difficult. They also provide some support when introducing these constructs, and gradual challenges in tackling them throughout the course of the game. However, support is not necessarily aimed at the specific difficulties occurring for particular constructs. For example, in Lighbot, support in the form of

a heuristic or rule is provided when the constructs of procedures and procedure calls are introduced; throughout the increasingly more difficult procedure exercises, however, no more support is provided on exactly the aspects of the procedure construct (e.g. recursive procedure calls) that make the exercise more difficult. **(++)** A few games deal with a variety of programming constructs, and thereby also focus on the particular difficulties of the construct in question. For example, Code Hunt focuses in detail on loops, conditionals and arrays, while providing extensive corrective feedback; Code Combat provides a wide variety of constructs, supported with explanations, examples and corrective feedback; Blockly makes use of various constructs and provides explicit transitions between easier and more difficult versions of a construct (e.g. in initial levels, code is represented as natural language, thereby providing a clear indication of its functionality, and replaced by programming syntax in subsequent levels).

Evaluation of program **(-)** We observed a few games in which no support for testing or debugging of programs is included. In these games, testing and debugging is only possible through running the program written by the learner. For example, Glitchspace requires learners to execute programs to move objects within the game world. Output of a program is shown in the behavior of the object; no further feedback on what a program is doing, and when it is doing certain actions, is provided. **(+)** Most games provide code highlighting, i.e. when a program is running, the corresponding code is highlighted. In addition, learners are able to start, stop and reset their programs at any time; sometimes programs can be run very slowly (e.g. Blockly), or the game provides additional comments on what the program is doing during execution (e.g. Ruby Warrior). These functionalities provide a basis for being able to track and trace errors in a program. However, none of the games we analyzed actually elaborate on this part of the programming process. Testing and debugging are not really stimulated, and facilities to enhance the process are only partially provided. For example, although Code Hunt gives corrective feedback in each level, this feedback is often general and not always corresponding to the error that was made. In addition, the hints that are given towards a solution are always aimed at making local fixes, and never at a more global refactoring of code. As another example, Move the Turtle has no code highlighting (i.e., tracking and tracing is difficult), but it does provide suggestions towards solving problems; Kodable, in contrast, indicates the location of the error with the image of a 'bug', but it doesn't give any indications on the causes of errors. **(++)** None of the games provide extensive facilities to support the aforementioned issues.

Syntax **(-)** Almost half of the games make use of either visual symbol blocks or programming language syntax as is, i.e. without any underlying directives or support. For example, Ruby Warrior works with syntax (i.e. Ruby syntax) without support, except for a small API; RoboLogic works with blocks, on which code symbols are depicted (e.g. a block with an arrow symbol to represent the command 'walk'). **(+)** A few other games make use of visual code blocks, i.e. blocks with little code fragments on top. By using this combination of blocks and code, they provide a small link between the visual representation and the actual code. For example, Tynker lost in space has visual code blocks 'if' and 'repeat until', in which learners can add a condition. **(++)** A few games make use of 'supported' programming language syntax only, or they also work with visual code blocks, but with an explicit link between visual representation and code. For example, Code Combat makes use of syntax with live auto completion, smart behaviors, and indentation support; Blockly makes an explicit transition between a visual code representation and Javascript by providing visual code blocks in initial levels, blocks on which learners can write code snippets in subsequent levels, and finally actual Javascript writing in the final levels; in addition, in each level, the Javascript code corresponding to the learners' program is shown.

Knowledge **(-)** We observed that a few games provide neither computer programming knowledge nor knowledge from any other learning domain. For example, Coddy Luck and Robozzle both have gameplay aimed at learning new programming constructs; this learning goal is conveyed through practice (gameplay) only, i.e. although they include a tutorial with definitions of all the available commands, information on programming constructs, or on more theoretical programming-related knowledge is not included; Glitchspace requires the application of mathematical concepts to solve some of the programming problems, but it does not include knowledge of that learning domain. **(+)** Most games include a limited amount of knowledge, by providing rules or heuristics that explain the operation or functionality of particular programming constructs. These rules are, however, presented isolated, i.e. not embedded into an overarching and permanently available theoretical framework. For example, Tynker lost in space provides a short description of the functionality of a for-loop at the moment errors are made in the use of such a for-loop. **(++)** Although none of the games we analyzed actually provide extensive programming theory, we found a few games in which knowledge is available on a more global level, in the form of documentation, and in which rules and heuristics are integrated into a more encompassing overview of the functionalities of a construct. For example, Blockly provides documentation in some of its levels. This documentation contains programming knowledge (e.g. explanation of the required input and output for a while-loop as well as a description and example of its functionality) as well as mathematics knowledge (e.g. the concept of angles, types and use of numbers etc.); Code Combat has knowledge information available in different types: through video tutorials, programming APIs describing method functionalities, and a code book describing particular constructs in detail.

Mental model **(+)** We found that all games, in a way, provide support for developing mental models. The fact that a programming problem is embedded into (attributes of) a game, or merely the fact that a visual pseudo code is used to compose programs, gives rise to the appearance of some kind of –maybe unintended- mental model of what programming, or particular aspects of programming, is. For example, many games make use of a robot that has to be programmed to find its way in a maze. This concept can be considered as a metaphor of what a computer, and programming a computer, entails. However, none of the games we analyzed also frame or support the interpretation of these 'unintended' mental models, i.e., metaphors are not linked back to the original domain of computer programming.

Problem-solving strategies **(-)** The majority of games do not provide support of specific step-by-step problem-solving strategies. **(+)** We found some games, however, that make use of limited scaffolding, by providing support when introducing new types of problems, and –abruptly- diminishing the support afterwards. Some games also make use of rules as affordances, i.e. only code constructs required to solve the problem at hand are made available. The latter can be considered as a way to make learners see relevant versus redundant information, and, that way, identify specific strategies to solve a problem. For example, Cato's Hike has tutorial levels that make use of scaffolding support, thereby also providing only relevant code constructs, and greying-out the redundant ones. **(++)** A few games provide this scaffolding and step-by-step problem-solving in a structured and sustained way (e.g. Code Combat).

Mapping **(-)** The majority of games do not provide a link between problems in a physical reality and their implementation in programming code. **(+)** One game – Colobot – requires learners to create programs that are a model (e.g. simulating electronic circuits) or a part of a physical reality (e.g. controlling a car's engine). The latter examples illustrate a link between a physical reality and a program, not between a physical reality and actual code. **(++)** A few games put more focus on mapping problems. For example, Code Combat and Blockly provide problems for which learners have to make explicit transformations between the concept of a real-life action (e.g. moving) and the code to describe this action (e.g. implementing a movement with lines, circles, angles etc.).

Attitude **(-)** A few games are fully aimed at trial-and-error behavior, without additional support or encouragement to find support elsewhere (e.g. Robozzle). **(+)** Most games occasionally provide small hints towards solutions, but these are often general and not specifically adapted to the problem at hand, and not necessarily provided at the moment the problem occurs. Although these can guide learners' problem-solving process, most problems are still left to learners' trial-and-error behavior. For example, Cargo-Bot provides one hint, usually in the form of (parts of) an algorithm description, in each level. If learners experience problem-solving difficulties, they can only rely on that particular hint. **(++)** A few games provide extensive corrective feedback and support, which can control and guide trial-and-error behavior. For example, Code Combat has links to sources of support directly available within its user interface, e.g. video tutorials and API's; Kodable is embedded into a broader learning platform, which provides extensive support.

Motivation **(-)** All the games we observed present some kind of initiating event or playful goal. However, in Code Hunt and Robozzle, these initiating events and goals are restricted to 'write or compose a program, and defeat your opponent with this program'. These games provide no narrative or fantasy, their challenges remain within the domain of programming only (e.g. a plain code editor in Code Hunt, aimed at programming functionalities only). **(+)** Most games encompass a program-your-robot concept, where the main goal is to program a robot to walk through a simple maze; no additional fantasy or more global narrative is provided. For example, RoboCom requires learners to program a robot machine to reach a target and avoid obstacles on the way; Cato's Hike makes use of the same straightforward program-your-robot concept, but encapsulates it into a more playful environment and story, i.e. a child game character has to move through a colorful garden to find its friend, and avoid bugs on the way. **(++)** A few games have far more elaborated stories and fantasies, or goals that closely adhere to young learners' world and interests, affording to immerse learners into a game world outside of the programming environment. For example, in Glitchspace, a first-person perspective is taken to stimulate immersion into a strange, dark city with moving objects; Coddy Luck requires learners to make a drawing, using plasticine-like pencil and eraser blocks.

PROBLEM SPECIFICITY AND GAME CHARACTERISTICS

We integrated the findings of both studies to gain insight into the mutual relationships between problem categories and how they are addressed within the programming games. This overall view is visualized in Figure 1. In the figure, the individual score of each game on a particular problem category (from Study 2) is converted to grey scales (cells with '-' scores are white; '+' scores are light grey, '++' scores are dark grey), and both problem categories and games are ranked according to their total scores (i.e. the total count of the number of +'s in each row and column). In what follows, patterns in the ranking of problem categories and games are discussed, based on the specificity of problem categories and their clustering into three main component groups (from Study 1).

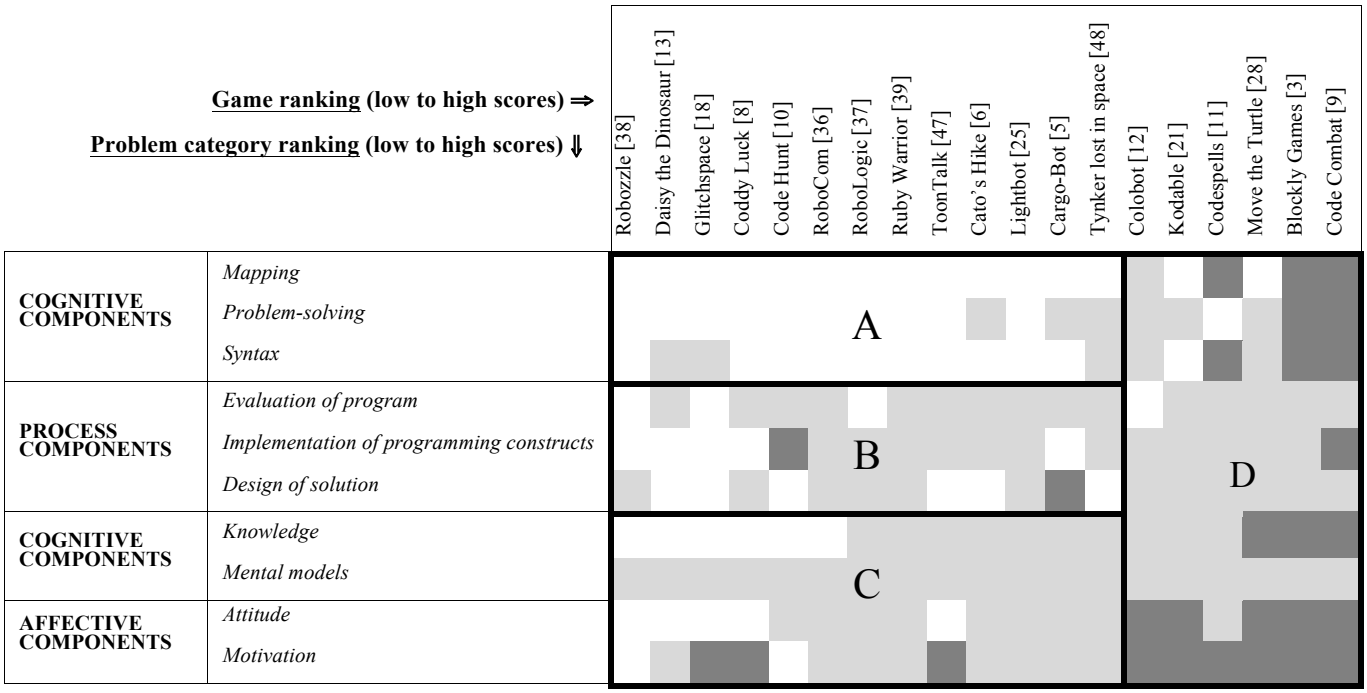

Figure 1. Programming games addressing problem categories. The table cell colors represent the score of a particular game on a particular problem category. Table rows and columns represent the ranking of problem categories and games respectively, both based on total scores. Clusters of scores are delineated with thick black borders and described with a letter (A, B, C, D).

First, the problem categories *mapping*, *problem-solving* and *syntax*, all identified as **cognitive components** supporting the programming process (Study 1), are situated at the lower end of the ranking (Figure 1); the problem categories *design*, *implementation* and *evaluation*, all identified as **process components** (Study 1), are situated towards the middle (Figure 1). Mapping, problem-solving and syntax problems all refer to difficulties that are specific to programming, i.e. they are not – in the same way - an issue in other learning domains. Process component problems (design, implementation and evaluation), on the other hand, are partly inferential from general learning principles and, therefore, applicable to any learning domain. For example, tackling a problem in any learning domain would require learners to apply some kind of planning or design before implementing a solution. It appears that the latter problems, related to the basic, overt parts of the programming process, are addressed more frequently, and without involving the more latent skills prerequisite to starting a programming process. They thereby focus on the general interpretation of a learning problem only, and not the programming-specific issues within. For example, many games address the implementation of a variety of programming constructs on a general level (zone B in Figure 1), but they thereby do not focus on the typical problems with the acquisition of preliminary skills such as writing correct syntax and using adequate problem-solving strategies (zone A in Figure 1).

Second, the problem categories *knowledge* and *mental models*, identified as **cognitive components** (Study 1), and the problem categories *attitude* and *motivation* (Study 1), identified as **affective components** (Study 1) are situated towards the higher end of the ranking (Figure 1). Many games moderately address the whole of the problems in these categories (zone C). These categories – especially mental models, attitude and motivation - describe difficulties for which a link with typical game characteristics that could provide support to the problem, is immediately clear. For example, fantasy elements typical of a game could easily be considered as natural support for developing mental models; narratives and events to increase motivation are readily available in games; the same goes for feedback, typical of games, as a means to influence trial-and-error attitudes and behavior. The contrast between the cognitive and affective components in zone C and the cognitive components in zone A could be explained by the affordances of these game characteristics. Indeed, the problem categories in zone A contain difficulties with, at first sight, little inherent connection points to the characteristics of a game. For example, for syntax problems, there is no directly visible link with attributes of a game that could self-evidently support the problem.

Finally, whereas the above considerations also apply to the games in zone D, they do so to a far lesser extent. The visualization of scores in this zone reveals that a high focus on various problem categories is clustered within a small group of games. These games also address process components on a general level, but they thereby put an explicit focus on specific cognitive component problems (mapping, problem-solving, syntax). In addition, they make

ample use of game affordances to address affective component problems (attitude, motivation).

DISCUSSION

Implications for programming game design and research

From the studies we conducted, we learned which problems novice programmers experience, and how these problems are addressed in existing programming games. However, adopting a content-driven perspective as the evaluation criterion in a game review (i.e., the extent to which specific programming difficulties are addressed is tentatively considered as the norm against which games are evaluated), we also learned how mapping out a current situation could reveal both gaps and opportunities in a current situation, thereby informing future game design and research.

To start with, we found that the actual content-related focus in the games is on basic design, implementation and evaluation of specific programming constructs, addressed from a general learning perspective. In conveying the learning content, games make use of some of their typical characteristics to provide motivation, feedback and mental models to the learner. For more programming-specific difficulties, little support is given. Instead, programming problems are integrated into games' typical challenge structure: a brief introductory support, followed by a set of increasingly more complex levels, for which support is no longer provided. At the point where learners would encounter typical difficulties, much is left to their trial-and-error behavior. However, trial-and-error behavior in itself was identified as a problem of novice programmers (Study 1): they have difficulties with searching for the cause of errors and with finding their own supportive information. Future research should therefore explore how trial-and-error behavior, although typically provoked through game play, could occur in a more controlled and supported fashion.

Second, attitude, motivation and mental model problems were considered as naturally linked to the affordances of specific games' attributes. However, game design should deploy games' characteristics in a way that does not ignore or even counter benefit the support for a specific programming problem. For example, trial-and-error behavior should not be over stimulated and should be supported with the proper feedback; fantasy and immersion elements, when unsupported, could easily cause misconceptions in the development of mental models.

Third, most specific programming problems could not be directly connected to characteristics or attributes of the games we investigated. We argue that the potential of typical game characteristics (such as the exploration of simulated environments, the extensive opportunities to repeat and practice specific issues, the adaptability of challenges, etc.) to support more specific problems should be further explored. For example, the design of learning mechanics (as suggested by [33]), translated for specific programming issues, could serve as inspiration for this kind of research. In addition, the design of such combinations of specific game attributes and specific learning content could be coupled to an investigation of their learning efficiency.

Fourth, some programming games (e.g. Daisy the Dinosaur) are aimed at very young children, only intended to offer them a grasp of the very basics of programming. Given this intention, it is evident that specific programming difficulties, such as recognizing loop boundaries or debugging recursion problems, are not an issue in the design of such games. Similarly, the platform for which a game has been developed can cause restrictions to the design affordances of that game (e.g. it would be inappropriate to provide extensive theory in a mobile game). Although this work did not take into account such age and platform restrictions, the aforementioned examples yield questions on where to position these programming games within the wider frame of a learning platform or curriculum, and, hence, on which context they could and should be used in. Since computer programming is not (yet) widely introduced into schools' curriculum, programming games will, for now, mainly be used in home or leisure contexts. In such contexts, learners typically play educational games on their own or without much support of a human mentor [20]. They could therefore greatly benefit of in-game support that is explicitly focused on their specific problems with acquiring programming skills.

Limitations

The novice programmer problems we identified in the expert workshop (Study 1) are not (intended to be) exhaustive – our final problems list was based on experiences of a limited number of participants, collected in a limited time period. However, comparing our workshop results to previous programming problems literature revealed the same (types of) problems. In addition, the problems we identified in the workshop serve as instructive, sensitizing examples, representing the particularities of a specific learning content and, like that, they are aimed at exploring and illustrating the potential need for a content-driven game design focus. To gain a more extensive view on young novice programmer problems, which was outside the scope of this work, future research should make use of additional research methods (e.g. observations) to, for example, involve the child's perspective.

The problem categories we defined (Study 1) do not have equal granularities. It would have been possible to place problems that are now in a fine-grained category into a broader, more high-level category. For example, syntax problems might be (partly) due to an underlying attitude problem and could thus have been positioned in the latter problem category. However, we tried to find a balance between marking each atomic example of a problem as a category on the one hand, and placing all problems into one abstract 'programming problems' category on the other hand. We argue that this balance, as a mixture of general

component (process, cognitive, affective) and problem category designations complemented with specific problem descriptions, could provide a starting point for analyzing or designing learning material in a game, since it draws attention towards general learning categories as well as specific occurrences of problems therein.

We analyzed only a limited set of programming games (Study 2). In addition, some of them were alpha releases, so we did not take into account the potential of the finished versions of these games. However, our main attempt with the game review was not to make any claims on "good" versus "bad" games, but to identify gaps and opportunities based on a "problem-specific" evaluation criterion. We argue that analysing existing games, maybe especially unfinished versions, could give indications on the potential benefits and drawbacks of particular design perspectives and, like that, provide inspiration for future research.

Conclusion

We identified 10 categories of young novice programmer problems, and examined how these were addressed in a set of 19 existing programming games. We found that learning problems common to any learning domain are addressed more than learning problems typical of the specific programming content. Based on our results, and extrapolating them to educational game design in general, we adopt an atomistic position towards the departure point for design, as a complement to the common holistic perspective. Problems specific to a particular learning content could be taken into account when designing learning activities and support within games, by making use of the characteristics and design affordances of games. Such an extended design perspective could contribute to the development of games that provide support specifically adapted to the difficulties of young novice programmers.

ACKNOWLEDGMENTS

We would like to thank all participants of our workshop for their valuable contribution, and our colleagues at Mintlab for providing feedback on the first versions of this paper.

REFERENCES

1. Barnes, T., Powell, E., Chaffin, A., Lipford, H. (2008). Game2Learn: Improving the motivation of CS1 students, *GDCSE'08*, Miami, USA.
2. Barnes, T., Richter, H., Chaffin, A., Godwin, A., Powell, E., Ralph, T., Matthews, P., Jordan, H. (2007). Game2Learn: A study of games as tools for learning introductory programming concepts, *SIGCSE'07*, USA.
3. Blockly [Game]. Retrieved from https://blockly-games.appspot.com/?lang=nl.
4. Bonar, J. & Soloway, E. (1989). Preprogramming knowledge: A major source of misconceptions in novice programmers. In E. Soloway & J.H. Spohrer (Eds.), *Studying the novice programmer*, 324-353, Hillsdale, NJ: Lawrence Erlbaum.
5. Cargo-Bot [Game]. (n.d.). Retrieved from https://itunes.apple.com/us/app/cargo-bot.
6. Cato's Hike [Game]. (n.d.). Retrieved from http://hwahba.com/catoshike.
7. Chaffin, A., Doran, K., Hicks, D., & Barnes, T. (2009). Experimental evaluation of teaching recursion in a video game, *Sandbox 2009*, New Orleans, Louisiana.
8. Coddy Luck [Game]. Retrieved from http://appike.com/coddy.
9. Code Combat [Game]. Retrieved from http://codecombat.com.
10. Code Hunt [Game]. (n.d.). Retrieved from https://www.codehunt.com/about.aspx.
11. Codespells [Game]. Retrieved from http://www.tucows.com/preview/1590538/CodeSpells.
12. Colobot [Game]. Retrieved from http://colobot.info/.
13. Daisy the Dinosaur [Game]. (n.d.). Retrieved from https://itunes.apple.com/us/app/daisy-the-dinosaur/id490514278?mt=8.
14. Echeverria, A., Barrios, E., Nussbaum, M., Amestica, M., Leclerc, S. (2012). The atomic intrinsic integration approach: a structured methodology for the design of games for the conceptual understanding of physics, *Computers & Education*, *59*, 806-816.
15. Esper, S., Wood, S.R., Foster, S.R., Lerner, S., Griswold, W.G. (2014). Codespells: How to design quests to teach Java concepts, *Journal of Computing Sciences in Colleges, 29*, 4, 114-122.
16. Gee, J.P. (2003). *What video games have to teach us about learning and literacy*. New York: Palgrave MacMillan.
17. Ginat, D. (2004). On novice loop boundaries and range conceptions. *Computer Science Education, 14*(3), 165-181.
18. Glitchspace [Game]. Retrieved from http://glitchspace.com/.
19. Kazimoglu, C., Kiernan, M., Bacon, L., MacKinnon, L. (2012). Learning programming at the computational thinking level via digital game-play. *Procedia Computer Science, 9*, 522-531.
20. Kerawalla, L., & Crook, C. (2002). Children's computer use at home and at school: context and continuity. *British Educational Research Journal, 28*, 752-771.
21. Kodable [Game]. Retrieved from http://www.surfscore.com.
22. Kumar, B. (2012). Gamification in education: Learn computer programming with fun. *International Journal of Computers and Distributed Systems*, 2, 1.

23. Lahtinen, E., Ala-Mutka, K., Järvinen, H-M (2005). A study of the difficulties of novice programmers, *ITiCSE'05*, Portugal.
24. Lieberman, D.A. (2006). What can we learn from playing interactive games? In P. Vorderer & J. Bryant (Eds.), *Playing Video games: motives, responses, and consequences*. Mahwah, NJ: Lawrence Erlbaum Associates.
25. Lightbot [Game]. Retrieved from http://www.lightbot.com.
26. Linn, M.C., & Dalbey, J. (1989). Cognitive consequences of programming instruction. In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer*, 129-159. Hillsdale, NJ: Lawrence Erlbaum.
27. Milne, I., Rowe, G. (2002). Difficulties in learning and teaching programming – views of students and tutors, *Journal of Education and Information Technologies*, 7:1, 55-66.
28. Move the Turtle [Game]. Retrieved from http://movetheturtle.com/.
29. Muratet, M., Torguet, P., Jessel, J-P, Viallet, F. (2009). Towards a serious game to help students learn computer programming, *International Journal of Computer Games Technology, vol. 2009,* Article ID 470590, 12 pages.
30. Muratet, M., Torguet, P., Viallet, F., & Jessel, J.P. (2011). Experimental feedback on Prog&Play: A Serious Game for Programming Practice, *Computer Graphics Forum, 30*, 1, pp. 61-73.
31. Perkins, D.N., & Martin, F. (1986). Fragile knowledge and neglected strategies in novice programmers. In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers, First Workshop*, 213-229, Norwood, NJ: Ablex.
32. Piteira, M. & Costa, C. (2012). Computer programming and novice programmers, *ISDOC'12*, Lisbon, Portugal.
33. Plass, J.L. (2011). Learning mechanics and assessment mechanics for games for learning, White paper-Institute for Games for Learning, Retrieved from http://g4li.org/wp-content/uploads/2011/11/G4LI-White-Paper-01-2011-Learning-Assessment-Mechanics.pdf.
34. Prensky, M. Computer games and learning: digital game-based learning (2011). In *Handbook of Computer Game Studies*, The MIT Press.
35. Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: a review and discussion. *Computer Science Education, 13*(2), 137-172.
36. RoboCom [Game]. Retrieved from https://itunes.apple.com/gb/app/robocombasic/id566959802?mt=8.
37. RoboLogic [Game]. Retrieved from http://www.digitalsirup.com/apps/app_robologic.html.
38. Robozzle [Game]. Retrieved from http://robozzle.com.
39. Ruby Warrior [Game]. Retrieved from https://www.bloc.io/ruby-warrior.
40. Sheehan, R. (2003). Children's perception of computer programming as an aid to designing programming environments, *IDC2003*, Preston, UK.
41. Shein, E. (2014). Should everybody learn to code? *Communications of the ACM*, *57*, 2.
42. Sherry, J. & Pacheco, A. (2006). Matching computer game genres to educational outcomes. *The Electronic journal of communication, 16*, 1-2.
43. Soloway, E. & Spohrer, J.C. (Eds.) (1989). *Studying the novice programmer*. Hillsdale, NL: Lawrence Erlbaum.
44. Squire, K. & Barab, B. (2004). Replaying history: Engaging urban underserved students in learning world history through computer simulation games. In *Proc. of ICLS,* Santa Monica, 505-512.
45. Sung, K. (2009). Computer games and traditional CS courses. *Communications of the ACM*, *52*, 12.
46. Tan, P., Ting, C., & Ling, S. (2009). Learning difficulties in programming courses: undergraduates' perspective and perception, *Proc. of ICCTD, 1*, 42-46.
47. ToonTalk [Game]. Retrieved from http://www.toontalk.com/.
48. Tynker lost in space [Game]. Retrieved from http://www.brainpop.com/games/tynkerlostinspace
49. Vahldick, A., Mendes, A.J., & Marcelino, M.J. (2014). A review of games designed to improve introductory computer programming competencies. *Frontiers in Education Conference*, 2014, IEEE, Madrid.
50. Wing, J.M. (2006). Computational Thinking, *Communications of the ACM, 49*(2), 33-35.
51. Wright, G.A., Rich, P., & Leatham, K.R. (2012). How programming fits with technology education curriculum, *Technology and Teaching Engineer, 71(7)*, 3-9.